

Beating the Productivity Checker Using Embedded Languages

Nils Anders Danielsson
University of Nottingham

Some total languages, like Agda and Coq, allow the use of guarded corecursion to construct infinite values and proofs. Guarded corecursion is a form of recursion in which arbitrary recursive calls are allowed, as long as they are guarded by a coinductive constructor. Guardedness ensures that programs are productive, i.e. that every finite prefix of an infinite value can be computed in finite time. However, many productive programs are not guarded, and it can be nontrivial to put them in guarded form.

This paper gives a method for turning a productive program into a guarded program. The method amounts to defining a problem-specific language as a data type, writing the program in the problem-specific language, and writing a guarded interpreter for this language.

1 Introduction

When working with infinite values in a total setting it is common to require that every value is *productive* (Sijtsma 1989): even though a value is conceptually infinite, it should always be possible to compute the next unit of information in finite time. The primitive methods for defining infinite values in the proof assistants Agda and Coq are based on *guarded corecursion* (Coquand 1994), which is a conservative approximation of productivity for coinductive types. The basic idea of guarded corecursion is that “corecursive calls” may only take place under guarding constructors, thus ensuring that the next unit of information—the next constructor—can always be computed. For instance, consider the following definition of $nats_{\geq n}$, the stream of successive natural numbers greater than or equal to n ($_{::}$ is the cons constructor for streams):

$$\begin{aligned} nats_{\geq} &: \mathbb{N} \rightarrow Stream\ \mathbb{N} \\ nats_{\geq} n &= n :: nats_{\geq} (suc\ n) \end{aligned}$$

This definition is guarded, and has the property that the next natural number can always be computed in finite time. As another example, consider *bad*:

$$\begin{aligned} bad &: Stream\ \mathbb{N} \\ bad &= tail\ (zero :: bad) \end{aligned}$$

This “definition” is not guarded (due to the presence of *tail*), nor is it productive: *bad* is not well-defined. Finally consider the following definition of the stream of natural numbers:

$$\begin{aligned} nats &: Stream\ \mathbb{N} \\ nats &= zero :: map\ suc\ nats \end{aligned}$$

This definition is productive, but unfortunately it is not guarded, because *map* is not a constructor. In fact, many productive definitions are not guarded, and it can be nontrivial to find equivalent guarded definitions.

The main contribution of this paper is a technique for translating a large class of productive but unguarded definitions into guarded definitions. The basic observation of the technique is that many productive definitions would be guarded if some functions were actually constructors. For instance, if *map* were a constructor, then *nats* would be guarded. The technique then amounts to defining a problem-specific language as a data type which includes a constructor for every function like *map*, implementing the productive definitions in a guarded way using this language, and implementing a guarded interpreter for the language. Optionally one can also prove that the resulting definitions satisfy their intended defining equations, and that these equations have unique solutions.

The technique relies on the use of data types defined using mixed induction and coinduction (see Section 2), so it requires a programming language with support for such definitions. The examples in the paper have been implemented using Agda (Norell 2007; Agda Team 2010), a dependently typed, total¹ functional programming language with good support for mixed induction and coinduction. The supporting source code is available to download (Danielsson 2010a).

Before we continue it may be useful to state some things which are *not* addressed by the paper:

- The paper’s focus is on establishing productivity, not on representing non-productive definitions, nor on making non-productive definitions total by restricting their types (Bertot 2005).
- No attempt is made to automate the technique: as it stands it provides a manual, somewhat ad hoc method for getting productive definitions accepted by a system based on guarded corecursion.

The rest of the paper is structured as follows: Section 2 discusses induction and coinduction in the context of Agda, Sections 3–8 (as well as Appendix A) introduce the language-based approach to productivity through a number of examples, Section 9 discusses related work, and Section 10 concludes.

2 Mixed Induction and Coinduction

This section gives a quick introduction to Agda, in particular to its support for mixed induction and coinduction. For more details, see Danielsson and Altenkirch (2010, Section 2).

In Agda the type of infinite streams can be defined as follows:

```
data Stream (A : Set) : Set where
  _::_ : A → ∞ (Stream A) → Stream A
```

This definition states that *Stream A* is a *Set* (“type”) with a single (infix) constructor *_::_* of type $A \rightarrow \infty (\text{Stream } A) \rightarrow \text{Stream } A$. The inclusion of ∞ in the type of *_::_* makes *Stream A* coinductive; without it the type would be empty. You should read $\infty (\text{Stream } A)$ as “delayed stream of As”—the function $\infty : \text{Set} \rightarrow \text{Set}$ is analogous to the suspension type constructors which are sometimes used to introduce non-strictness in strict languages (Wadler et al. 1998), and closely related to the domain-theoretic notion of lifting. However, Agda programs are required to be total.

We can construct infinite values by guarded corecursion. For instance, we can define a function which combines two streams in a pointwise manner as follows:²

```
zipWith : {A B C : Set} → (A → B → C) → Stream A → Stream B → Stream C
zipWith f (x :: xs) (y :: ys) = f x y :: # zipWith f (xs) (ys)
```

¹Agda is an experimental system with neither a formalised meta-theory nor a verified type checker, so take words such as “total” with a grain of salt.

²The notation $\{A B C : \text{Set}\} \rightarrow \dots$ means that *zipWith* takes three *implicit* arguments *A*, *B* and *C*, all of type *Set*. These arguments do not need to be given explicitly if Agda can infer them.

This definition uses the coinductive delay constructor $\sharp_$ (sharp)³ and the force function \flat (flat):

$$\begin{aligned}\sharp_ &: \{A : \text{Set}\} \rightarrow A \rightarrow \infty A \\ \flat &: \{A : \text{Set}\} \rightarrow \infty A \rightarrow A\end{aligned}$$

Agda views *zipWith* as guarded, because there is no non-constructor function between the left-hand side and the corecursive call, and there is at least one use of the guarding coinductive constructor $\sharp_$. This constructor has special status: it is treated as a constructor by Agda’s productivity checker, but may not be used in patterns. Instead one can use the force function: $\flat (\sharp x)$ reduces to x .

As another example, consider the following definition of equality—bisimilarity—for streams (which makes use of the fact that constructors can be overloaded):

$$\begin{aligned}\mathbf{data} _ \approx _ &: \{A : \text{Set}\} : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set} \mathbf{where} \\ _ :: _ &: (x : A) \rightarrow \{xs\ ys : \infty (\text{Stream } A)\} \rightarrow \infty (\flat xs \approx \flat ys) \rightarrow x :: xs \approx x :: ys\end{aligned}$$

This definition states that two streams are equal if their heads are identical and their tails are equal (coinductively). Note that the elements of this type are equality *proofs*. We can establish equalities by constructing proofs using guarded corecursion. For instance, we can prove symmetry as follows:

$$\begin{aligned}\text{sym} &: \{A : \text{Set}\} \rightarrow \{xs\ ys : \text{Stream } A\} \rightarrow xs \approx ys \rightarrow ys \approx xs \\ \text{sym} (x :: xs \approx ys) &= x :: \sharp \text{sym} (\flat xs \approx \flat ys)\end{aligned}$$

(Note that $xs \approx ys$ is an ordinary variable, albeit perhaps with an unusual name.)

Let us now consider a definition which uses both induction and coinduction. The type $SP\ A\ B$ of stream processors (Hancock et al. 2009)—representations of programs taking streams of A s to streams of B s—can be defined as follows:

$$\begin{aligned}\mathbf{data} SP\ (A\ B : \text{Set}) &: \text{Set} \mathbf{where} \\ \text{put} &: B \rightarrow \infty (SP\ A\ B) \rightarrow SP\ A\ B \\ \text{get} &: (A \rightarrow SP\ A\ B) \rightarrow SP\ A\ B\end{aligned}$$

Here $\text{put } b\ sp$ is intended to output b and continue with sp , while $\text{get } f$ is intended to read an element a and continue with $f\ a$. You can see the type as the nested fixpoint⁴ $\nu X. \mu Y. B \times X + (A \rightarrow Y)$ —in fact, *all* (non-mutual) data types in the paper can be seen as nested fixpoints of the form $\nu X. \mu Y. F\ X\ Y$ (and mutually defined data types can be merged by adding an index). Note that the recursive argument of put is delayed (coinductive), whereas the recursive argument of get is not. This means that we can have an infinite number of consecutive put constructors, but only a finite number of consecutive get s; definitions such as the following one are not guarded and not accepted:

$$\begin{aligned}\text{sink} &: \{A\ B : \text{Set}\} \rightarrow SP\ A\ B \\ \text{sink} &= \text{get } (\lambda _ \rightarrow \text{sink})\end{aligned}$$

The definition of *sink* is not problematic in and of itself (assuming that it is not evaluated too eagerly). However, by ruling out such definitions we make other definitions possible, for instance the following one, which gives the semantics of a stream processor:

³The prefix operator $\sharp_$ is the most tightly binding operator in this paper; ordinary function application binds tighter, though.

⁴Currently this is not quite correct in Agda (Altenkirch and Danielsson 2010), but for the purposes of this paper the differences are irrelevant.

$$\begin{aligned}
\llbracket _ \rrbracket &: \{A\ B : \text{Set}\} \rightarrow SP\ A\ B \rightarrow \text{Stream}\ A \rightarrow \text{Stream}\ B \\
\llbracket \text{put } b\ sp \rrbracket as &= b :: \# (\llbracket \text{ }^b sp \rrbracket as) \\
\llbracket \text{get } f \rrbracket (a :: as) &= \llbracket f\ a \rrbracket (\text{ }^b as)
\end{aligned}$$

This function is accepted by Agda because it is defined using a lexicographic combination of guarded corecursion and structural recursion. In this particular example the first component of the lexicographic product is the “guardedness”, and the second component is the *inductive* structure of the stream processor:

- In the first clause the corecursive call is guarded. The stream processor is not structurally smaller, due to the use of the force function (^b), but this is irrelevant.
- In the second clause the corecursive call is not guarded, but there is no non-constructor function between the left-hand side and the corecursive call, so we say that “guardedness is preserved”. On the other hand, the stream processor argument is strictly structurally smaller ($f\ x$ is smaller than $\text{get } f$ for any x).

Armed with the knowledge that there can only be a finite number of consecutive `get` constructors we conclude that, when evaluating $\llbracket sp \rrbracket as$, we must eventually reach the first clause. At this stage we can immediately inspect the head element of the output stream, because the second clause does not introduce any interfering destructors.

As a final example, consider *filter*, which is not accepted by Agda:

$$\begin{aligned}
\text{filter} &: \{A : \text{Set}\} \rightarrow (A \rightarrow \text{Bool}) \rightarrow \text{Stream}\ A \rightarrow \text{Stream}\ A \\
\text{filter } p\ (x :: xs) &\textbf{with } p\ x \\
\text{filter } p\ (x :: xs) \mid \text{true} &= x :: \# \text{filter } p\ (\text{ }^b xs) \\
\text{filter } p\ (x :: xs) \mid \text{false} &= \text{filter } p\ (\text{ }^b xs)
\end{aligned}$$

(Here the **with** construct is used to pattern match on $p\ x$.) The first corecursive call is guarded, but in the last clause the call is not guarded, and nothing is structurally smaller, so this function is not accepted.

The explanations above should suffice to understand the definitions in this paper—in fact, most definitions use less complicated recursion principles than the one used by $\llbracket _ \rrbracket$. For more information about Agda’s criterion for accepting a function as total, see Danielsson and Altenkirch (2010, Section 2.5).

Before we continue note that, in order to reduce clutter, the declarations of implicit arguments have been omitted in the remainder of the paper.

3 Making Programs Guarded

As noted in the introduction guardedness is sometimes an inconvenient restriction: there are productive programs which are not syntactically guarded. This section introduces a language-based technique for putting definitions in guarded form.

Consider the following definition of the stream of Fibonacci numbers:

$$\begin{aligned}
\text{fib} &: \text{Stream}\ \mathbb{N} \\
\text{fib} &= 0 :: \# \text{zipWith } _+ _ \text{fib}\ (1 :: \# \text{fib})
\end{aligned}$$

While the definition of *fib* is productive, it is not guarded, because the function *zipWith* is not a constructor. If *zipWith* were a constructor the definition would be guarded, though, and this presents a way out:

we can define a problem-specific language which includes *zipWith* as a constructor, and then define an interpreter for the language by using guarded corecursion.

A simple language of stream programs can be defined as follows:⁵

data $Stream_P : Set \rightarrow Set_1$ **where**
 $_::_ : A \rightarrow \infty (Stream_P A) \rightarrow Stream_P A$
 $zipWith : (A \rightarrow B \rightarrow C) \rightarrow Stream_P A \rightarrow Stream_P B \rightarrow Stream_P C$

Note that the stream program argument of $_::_$ is coinductive, while the arguments of *zipWith* are inductive; a stream program consisting of an infinitely deep application of *zipWith*s would not be productive.

Stream programs will be turned into streams in two steps. First a kind of weak head normal form (WHNF) for stream programs is computed recursively, and then the resulting stream is computed corecursively. The WHNFs are defined in the following way:

data $Stream_W : Set \rightarrow Set_1$ **where**
 $_::_ : A \rightarrow Stream_P A \rightarrow Stream_W A$

Note that the stream argument to $_::_$ is a (“suspended”) program, not a WHNF. The function *whnf* which computes WHNFs can be defined by structural recursion:

$whnf : Stream_P A \rightarrow Stream_W A$
 $whnf (x :: xs) = x ::^b xs$
 $whnf (zipWith f xs ys) = zipWith_W f (whnf xs) (whnf ys)$

Here $zipWith_W$ is defined by simple case analysis:

$zipWith_W : (A \rightarrow B \rightarrow C) \rightarrow Stream_W A \rightarrow Stream_W B \rightarrow Stream_W C$
 $zipWith_W f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys$

WHNFs can then be turned into streams corecursively:

mutual
 $\llbracket _ \rrbracket_W : Stream_W A \rightarrow Stream A$
 $\llbracket x :: xs \rrbracket_W = x ::^\# \llbracket xs \rrbracket_P$
 $\llbracket _ \rrbracket_P : Stream_P A \rightarrow Stream A$
 $\llbracket xs \rrbracket_P = \llbracket whnf xs \rrbracket_W$

Note that this definition is guarded. (Agda accepts definitions like this one even though it is split up over two mutually defined functions; alternatively one could write $\llbracket x :: xs \rrbracket_W = x ::^\# \llbracket whnf xs \rrbracket_W$ and define $\llbracket _ \rrbracket_P$ separately.)

Given the language above we can now define the stream of Fibonacci numbers using guarded corecursion:

$fib_P : Stream_P \mathbb{N}$ $fib : Stream \mathbb{N}$
 $fib_P = 0 ::^\# zipWith _+_ fib_P (1 ::^\# fib_P)$ $fib = \llbracket fib_P \rrbracket_P$

⁵ Set_1 is a type of large types; ∞ has type $Set_i \rightarrow Set_i$ for any i .

One can prove that this definition satisfies the original equation for *fib* by first proving corecursively that $\llbracket - \rrbracket_P$ is homomorphic with respect to *zipWith*/*zipWith*:

$$\begin{aligned} \text{zipWith-hom} &: (f : A \rightarrow B \rightarrow C) \rightarrow (xs : \text{Stream } A) \rightarrow (ys : \text{Stream } B) \rightarrow \\ &\quad \llbracket \text{zipWith } f \text{ } xs \text{ } ys \rrbracket_P \approx \text{zipWith } f \llbracket xs \rrbracket_P \llbracket ys \rrbracket_P \\ \text{fib-correct} &: \text{fib} \approx 0 :: \# \text{zipWith } - + - \text{fib } (1 :: \# \text{fib}) \end{aligned}$$

For the omitted proofs, see Danielsson (2010a). One may also want to establish that the original equation for *fib* defines the stream completely, i.e. that it has a *unique* solution. For an explanation of how this can be done, see Section 5.

It can be instructive to see what would happen if we tried to use the method above to implement the ill-defined stream *bad* from the introduction. Defining the language and giving a “definition” for *bad* is straightforward:

$$\begin{aligned} \text{data Stream}_P (A : \text{Set}) : \text{Set where} & & \text{bad} : \text{Stream}_P \mathbb{N} \\ _::_ : A \rightarrow \infty (\text{Stream}_P A) \rightarrow \text{Stream}_P A & & \text{bad} = \text{tail } (\text{zero} :: \# \text{bad}) \\ \text{tail} : \text{Stream}_P A \rightarrow \text{Stream}_P A & & \end{aligned}$$

However, turning stream programs into streams becomes tricky. How would *tail_W* be defined?

$$\begin{aligned} \text{data Stream}_W (A : \text{Set}) : \text{Set where} & & \text{tail}_W : \text{Stream}_W A \rightarrow \text{Stream}_W A \\ _::_ : A \rightarrow \text{Stream}_P A \rightarrow \text{Stream}_W A & & \text{tail}_W (x :: xs) = ? \end{aligned}$$

Note that, in the body of *tail_W*, *xs* is a stream program, but we need to produce a WHNF.

4 Several Types at Once

The technique introduced in Section 3 is not limited to streams. In fact, it can be used with several types at the same time. To illustrate how this can be done I will implement circular breadth-first labelling of trees à la Jones and Gibbons (1993).

The following type of potentially infinite binary trees will be used:

$$\begin{aligned} \text{data Tree } (A : \text{Set}) : \text{Set where} & \\ \text{leaf} &: \text{Tree } A \\ \text{node} &: \infty (\text{Tree } A) \rightarrow A \rightarrow \infty (\text{Tree } A) \rightarrow \text{Tree } A \end{aligned}$$

Jones and Gibbons’ implementation can be described as follows. First a labelling function *lab* is defined. This function takes a tree, along with a stream of streams of new labels. The labels in a prefix of the *n*-th stream are used to label the *n*-th level of the tree, and the remaining labels are returned from *lab*:

$$\begin{aligned} \text{lab} &: \text{Tree } A \rightarrow \text{Stream } (\text{Stream } B) \rightarrow \text{Tree } B \times \text{Stream } (\text{Stream } B) \\ \text{lab leaf} \quad \quad \text{bss} &= (\text{leaf}, \quad \quad \text{bss}) \\ \text{lab (node } l _ r) ((b :: bs) :: \text{bss}) &= (\text{node } (\# l') \text{ } b \text{ } (\# r'), \text{ } ^b bs :: \# \text{bss}'') \\ \text{where} & \\ (^{l'} \text{ }, \text{bss}') &= \text{lab } (^b l) (^b \text{bss}) \\ (^{r'} \text{ }, \text{bss}'') &= \text{lab } (^b r) \text{bss}' \end{aligned}$$

This code is not accepted by Agda, because the recursive calls are not guarded (their results are destructured, and furthermore *lab*, which is not a constructor, is applied to a part of one of the results).⁶ The next step in Jones and Gibbons’ implementation is to construct the stream of streams of labels which is used by *lab*, and use these streams to compute the relabelled tree. This is done using a circular construction:

```
label : Tree A → Stream B → Tree B
label t bs = t'
  where (t', bss) = lab t (bs :: # bss)
```

This code is not accepted by Agda, because *lab* is not a constructor, and furthermore the result of *lab* is destructured.

To implement breadth-first labelling in the style of Jones and Gibbons the following universe of trees, streams, products and arbitrary (small) types will be used:

data $U : Set_1$ where	$El : U \rightarrow Set$
tree : $U \rightarrow U$	$El (\text{tree } a) = Tree (El a)$
stream : $U \rightarrow U$	$El (\text{stream } a) = Stream (El a)$
$_{\otimes}$: $U \rightarrow U \rightarrow U$	$El (a \otimes b) = El a \times El b$
$[_]$: $Set \rightarrow U$	$El [A] = A$

The type U defines codes for elements of the universe, and El interprets these codes.

By indexing the program and WHNF types by codes from the universe U we can work with several types at once:

```
mutual
data  $El_P : U \rightarrow Set_1$  where
  ↓ :  $El_W a \rightarrow El_P a$ 
  fst :  $El_P (a \otimes b) \rightarrow El_P a$ 
  snd :  $El_P (a \otimes b) \rightarrow El_P b$ 
  lab :  $Tree A \rightarrow El_P (\text{stream } [Stream B]) \rightarrow El_P (\text{tree } [B] \otimes \text{stream } [Stream B])$ 

data  $El_W : U \rightarrow Set_1$  where
  leaf :  $El_W (\text{tree } a)$ 
  node :  $\infty (El_P (\text{tree } a)) \rightarrow El_W a \rightarrow \infty (El_P (\text{tree } a)) \rightarrow El_W (\text{tree } a)$ 
  _::_ :  $El_W a \rightarrow \infty (El_P (\text{stream } a)) \rightarrow El_W (\text{stream } a)$ 
  _,-_ :  $El_W a \rightarrow El_W b \rightarrow El_W (a \otimes b)$ 
  [-] :  $A \rightarrow El_W [A]$ 
```

Note that only those constructor arguments which are delayed are represented as programs in the definition of El_W —the other arguments can be viewed as “strict”. Note also that, unlike in Section 3, the two types are defined mutually: the WHNF type is included in the type of programs using the constructor \downarrow . This makes the program type less usable (the term $\text{fst } p :: xs$ is not well-typed, for instance), but avoids some code duplication. An alternative would be to merge the definitions of El_P and El_W , and use an additional index to specify which programs are in weak head normal form.

The type of *lab* may seem a bit strange: the inner and outer streams are represented differently. One reason for this design choice can be seen in the definition of *lab_W*:

⁶Agda does not support pattern matching in where clauses as used here, but one can use projection functions instead.

$$\begin{aligned}
lab_W &: Tree\ A \rightarrow El_W (\text{stream } [\text{Stream } B]) \rightarrow El_W (\text{tree } [B] \otimes \text{stream } [\text{Stream } B]) \\
lab_W \text{ leaf } \quad \quad \quad bss &= (\text{leaf}, \quad \quad \quad bss) \\
lab_W (\text{node } l _ r) ([b :: bs] :: bss) &= (\text{node } (\# \text{fst } x) [b] (\# \text{fst } y), [^b bs] :: \# \text{snd } y) \\
\textbf{where} \\
x &= \text{lab } (^b l) (^b bss) \\
y &= \text{lab } (^b r) (\text{snd } x)
\end{aligned}$$

Consider the second clause. If lab_W had the type

$$Tree\ A \rightarrow El_W (\text{stream } (\text{stream } b)) \rightarrow El_W (\text{tree } b \otimes \text{stream } (\text{stream } b)),$$

then the analogue of bs would be a *program*, but the head of the resulting stream of streams ($[^b bs]$ in the definition above) must be a WHNF. The use of “raw” inner streams also means that the input to the *label* function does not need to be converted.

Note that lab_W is non-recursive. The remainder of *whnf* is straightforward to implement using structural recursion:

$$\begin{aligned}
fst_W &: El_W (a \otimes b) \rightarrow El_W a & whnf &: El_P a \rightarrow El_W a \\
fst_W (x, y) &= x & whnf (\downarrow w) &= w \\
snd_W &: El_W (a \otimes b) \rightarrow El_W b & whnf (\text{fst } p) &= fst_W (whnf p) \\
snd_W (x, y) &= y & whnf (\text{snd } p) &= snd_W (whnf p) \\
& & whnf (\text{lab } t \ bss) &= lab_W t (whnf bss)
\end{aligned}$$

It is also easy to define $\llbracket - \rrbracket_W$ and $\llbracket - \rrbracket_P$. These definitions use a lexicographic combination of guarded corecursion and structural recursion (see Section 2):

$$\begin{aligned}
\textbf{mutual} \\
\llbracket - \rrbracket_W &: El_W a \rightarrow El\ a \\
\llbracket \text{leaf} \rrbracket_W &= \text{leaf} \\
\llbracket \text{node } l \ x \ r \rrbracket_W &= \text{node } (\# \llbracket ^b l \rrbracket_P) (\llbracket x \rrbracket_W) (\# \llbracket ^b r \rrbracket_P) \\
\llbracket x :: xs \rrbracket_W &= \llbracket x \rrbracket_W :: \# \llbracket ^b xs \rrbracket_P \\
\llbracket (x, y) \rrbracket_W &= (\llbracket x \rrbracket_W, \llbracket y \rrbracket_W) \\
\llbracket [x] \rrbracket_W &= x \\
\llbracket - \rrbracket_P &: El_P a \rightarrow El\ a \\
\llbracket p \rrbracket_P &= \llbracket whnf\ p \rrbracket_W
\end{aligned}$$

Finally we can define *label*:

$$\begin{aligned}
label' &: Tree\ A \rightarrow Stream\ B \rightarrow El_P (\text{tree } [B] \otimes \text{stream } [\text{Stream } B]) \\
label' \ t \ bs &= \text{lab } t (\downarrow ([bs] :: \# \text{snd } (label' \ t \ bs))) \\
label &: Tree\ A \rightarrow Stream\ B \rightarrow Tree\ B \\
label \ t \ bs &= \llbracket \text{fst } (label' \ t \ bs) \rrbracket_P
\end{aligned}$$

Note that the helper function *label'*, which corresponds to the cyclic part of the original *label*, is defined using guarded corecursion.

I have proved that the definition of *label* is correct: the resulting tree has the same shape as the original one, and a breadth-first traversal of the resulting tree produces a potentially infinite list of labels

which is a prefix of the stream given to *label*. To state correctness I extended the universe with support for potentially infinite lists, and added some programs to the El_P type. For details of the statement and proof, see Danielsson (2010a).

5 Making Proofs Guarded

The language-based approach to guardedness introduced in Section 3 has some problems when applied to programs:

- The interpretive overhead, compared to a direct implementation, can be substantial. For instance, computing the n -th element of the stream *fib* defined in Section 3 requires a number of additions which is exponential in n , whereas if $\text{fib} = 0 :: \# \text{zipWith } _ + _ \text{ fib } (1 :: \# \text{fib})$ is implemented directly in a language which uses call-by-need this computation only requires $\mathcal{O}(n)$ additions. The reason for this discrepancy is that the interpreter $\llbracket _ \rrbracket_P$ does not preserve sharing. One could perhaps work around this problem by writing a more complicated interpreter, but this seems counterproductive: why spend effort writing a new interpreter when one is already provided by the host programming language (or the underlying hardware)?
- Proving properties about the interpreted definitions (for instance to establish that they are correct) can be awkward, because this amounts to proving properties about the interpreter.

However, these problems are usually irrelevant for *proofs*: the run-time complexity of proofs is rarely important, and any proof of a property is usually as good as any other. Hence the approach is likely to be more useful for making proofs guarded, than for making programs guarded.

This section shows how the technique can be applied to proofs. Hinze (2008) advocates proving stream identities using a uniqueness property. One example in his paper is the iterate fusion law:

$$\begin{aligned} \text{fusion} : (h : A \rightarrow B) \rightarrow (f_1 : A \rightarrow A) \rightarrow (f_2 : B \rightarrow B) \rightarrow \\ ((x : A) \rightarrow h (f_1 x) \equiv f_2 (h x)) \rightarrow \\ (x : A) \rightarrow \text{map } h (\text{iterate } f_1 x) \approx \text{iterate } f_2 (h x) \end{aligned}$$

Here *map* and *iterate* are defined as follows:

$$\begin{aligned} \text{map} : (A \rightarrow B) \rightarrow \text{Stream } A \rightarrow \text{Stream } B & \quad \text{iterate} : (A \rightarrow A) \rightarrow A \rightarrow \text{Stream } A \\ \text{map } f (x :: xs) = f x :: \# \text{map } f (\flat xs) & \quad \text{iterate } f x = x :: \# \text{iterate } f (f x) \end{aligned}$$

Hinze proves the iterate fusion law by establishing that the left and right hand sides both satisfy the same guarded equation, $f x \approx h x :: \# f (f_1 x)$ (where f is the “unknown variable”):

$$\begin{aligned} \text{map } h (\text{iterate } f_1 x) & \quad \approx \langle \text{by definition} \rangle \\ h x :: \# \text{map } h (\text{iterate } f_1 (f_1 x)) & \\ \\ h x :: \# \text{iterate } f_2 (h (f_1 x)) & \quad \approx \langle \text{assumption} \rangle \\ h x :: \# \text{iterate } f_2 (f_2 (h x)) & \quad \approx \langle \text{by definition} \rangle \\ \text{iterate } f_2 (h x) & \end{aligned}$$

The separately proved⁷ fact that the equation has a unique solution then implies that $\text{map } h (\text{iterate } f_1 x)$ and $\text{iterate } f_2 (h x)$ are equal.

⁷Hinze proves this using a method described by Rutten (2003), which in fact is closely related to the method described here, see Section 9.

Note that the proof above is almost a proof by guarded coinduction: the two equational reasoning blocks can be joined by an application of the coinductive hypothesis. However, the second block uses transitivity, thus destroying guardedness. We can work around this problem by following the approach introduced in Section 3. Let us define a language of equality proof “programs” as follows:

data $\approx_P_- : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set}$ **where**
 $_::_ : (x : A) \rightarrow \infty ({}^b xs \approx_P {}^b ys) \rightarrow x :: xs \approx_P x :: ys$
 $\approx(_)_ : (xs : \text{Stream } A) \rightarrow xs \approx_P ys \rightarrow ys \approx_P zs \rightarrow xs \approx_P zs$
 $_ \square : (xs : \text{Stream } A) \rightarrow xs \approx_P xs$

The last two constructors represent transitivity and reflexivity, respectively. Note that the transitivity constructor is inductive; a coinductive transitivity constructor would make the relation trivial (see Danielsson and Altenkirch (2010)). The somewhat odd names were chosen to make the proof of the iterate fusion law more readable, following Norell (2007). Just remember that $\approx(_)_$ and $_ \square$ are both weakly binding, with $\approx(_)_$ right associative and binding weaker than $_ \square$:

$\text{fusion} : (h : A \rightarrow B) \rightarrow (f_1 : A \rightarrow A) \rightarrow (f_2 : B \rightarrow B) \rightarrow$
 $((x : A) \rightarrow h(f_1 x) \equiv f_2(h x)) \rightarrow$
 $(x : A) \rightarrow \text{map } h(\text{iterate } f_1 x) \approx_P \text{iterate } f_2(h x)$
 $\text{fusion } h f_1 f_2 \text{ hyp } x =$
 $\text{map } h(\text{iterate } f_1 x) \quad \approx(\text{by definition})$
 $h x :: \# \text{map } h(\text{iterate } f_1(f_1 x)) \approx(h x :: \# \text{fusion } h f_1 f_2 \text{ hyp } (f_1 x))$
 $h x :: \# \text{iterate } f_2(h(f_1 x)) \approx(h x :: \# \text{iterate-cong } f_2(\text{hyp } x))$
 $h x :: \# \text{iterate } f_2(f_2(h x)) \approx(\text{by definition})$
 $\text{iterate } f_2(h x) \quad \square$

Note that the definition of *fusion* is guarded. The definition uses some simple lemmas (*iterate-cong*, *by* and *definition*), which are omitted here.

In order to finish the proof of the iterate fusion law we have to show that \approx_P_- is sound with respect to \approx_- . To do this one can first define a type of WHNFs:

data $\approx_W_- : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set}$ **where**
 $_::_ : (x : A) \rightarrow {}^b xs \approx_P {}^b ys \rightarrow x :: xs \approx_W x :: ys$

It is easy to establish, by simple case analysis, that this relation is a preorder:

$\text{refl}_W : (xs : \text{Stream } A) \rightarrow xs \approx_W xs$
 $\text{trans}_W : xs \approx_W ys \rightarrow ys \approx_W zs \rightarrow xs \approx_W zs$

It follows by structural recursion that programs can be turned into WHNFs:

$\text{whnf} : xs \approx_P ys \rightarrow xs \approx_W ys$
 $\text{whnf } (x :: xs \approx_P ys) = x :: {}^b xs \approx_P ys$
 $\text{whnf } (xs \approx(_)_ ys \approx_P zs) = \text{trans}_W(\text{whnf } xs \approx_P ys)(\text{whnf } ys \approx_P zs)$
 $\text{whnf } (xs \square) = \text{refl}_W xs$

Finally soundness can be proved using guarded corecursion:

mutual

$$\begin{aligned}
& \text{sound}_W : xs \approx_W ys \rightarrow xs \approx ys \\
& \text{sound}_W (x :: xs \approx ys) = x :: \# \text{sound}_P xs \approx ys \\
& \text{sound}_P : xs \approx_P ys \rightarrow xs \approx ys \\
& \text{sound}_P xs \approx ys = \text{sound}_W (\text{whnf } xs \approx ys)
\end{aligned}$$

Note that there is no need to prove that the application $\text{sound}_P (\text{fusion } h f_1 f_2 \text{ hyp } x)$ satisfies its intended defining equation, whatever that would be, or that this equation has a unique solution.

Using the language-based approach to guardedness I have formalised a number of examples from Hinze’s paper, see Danielsson (2010a). Rephrasing the proofs using guarded coinduction turned out to be unproblematic.

As a further example, let us show that the defining equation for *fib* (see Section 3) has a unique solution. We can state the problem as follows:

$$\begin{aligned}
& \text{fib-rhs} : \text{Stream } \mathbb{N} \rightarrow \text{Stream } \mathbb{N} \\
& \text{fib-rhs } ns = 0 :: \# \text{zipWith } _+_ \text{ ns } (1 :: \# ns) \\
& \text{fib-unique} : (ms \text{ ns} : \text{Stream } \mathbb{N}) \rightarrow ms \approx \text{fib-rhs } ms \rightarrow ns \approx \text{fib-rhs } ns \rightarrow ms \approx_P ns
\end{aligned}$$

The type \approx_P used here is different from the one used above: the proof will make use of the congruence of *zipWith*, and the coinductive hypothesis will be an argument to this congruence, so a constructor for the congruence is included among the equality proof programs:

$$\begin{aligned}
& \text{data } \approx_P : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set} \text{ where} \\
& \dots \\
& \text{zipWith-cong} : (f : A \rightarrow A \rightarrow A) \rightarrow xs_1 \approx_P ys_1 \rightarrow xs_2 \approx_P ys_2 \rightarrow \\
& \quad \text{zipWith } f \text{ xs}_1 \text{ xs}_2 \approx_P \text{zipWith } f \text{ ys}_1 \text{ ys}_2
\end{aligned}$$

It is easy to extend the definition of *whnf* to support *zipWith-cong*, using which we can define *fib-unique* as follows:

$$\begin{aligned}
& \text{fib-unique } ms \text{ ns hyp}_1 \text{ hyp}_2 = \\
& \quad ms \approx \langle \text{complete}_P \text{ hyp}_1 \rangle \\
& \quad \text{fib-rhs } ms \approx \langle 0 :: \# \text{zipWith-cong } _+_ \text{ (fib-unique } ms \text{ ns hyp}_1 \text{ hyp}_2) \\
& \quad \quad \quad (1 :: \# \text{fib-unique } ms \text{ ns hyp}_1 \text{ hyp}_2) \rangle \\
& \quad \text{fib-rhs } ns \approx \langle \text{complete}_P (\text{sym hyp}_2) \rangle \\
& \quad ns \quad \square
\end{aligned}$$

Here *sym* is the proof of symmetry of \approx from Section 2, and *complete_P* shows that \approx_P is complete with respect to \approx :

$$\begin{aligned}
& \text{complete}_P : xs \approx ys \rightarrow xs \approx_P ys \\
& \text{complete}_P (x :: xs \approx ys) = x :: \# \text{complete}_P (\text{hyp } xs \approx ys)
\end{aligned}$$

6 Destructors

The following, alternative definition of the Fibonacci sequence is not directly supported by the framework outlined in previous sections:

$$\begin{aligned} fib &: Stream \mathbb{N} \\ fib &= 0 :: \# (1 :: \# zipWith _+ _ fib (tail fib)) \end{aligned}$$

The problem is the use of the destructor *tail*. Unrestricted use of destructors can lead to non-productive “definitions”, as demonstrated by *bad* (see Section 1). However, destructors can be incorporated by extending the program type with an index which indicates when they can be used.

Consider the following type of stream programs:

$$\begin{aligned} \text{data } Stream_P &: Bool \rightarrow Set \rightarrow Set_1 \text{ where} \\ [_] &: \infty (Stream_P \text{ true } A) \rightarrow Stream_P \text{ false } A \\ _::_ &: A \rightarrow Stream_P \text{ false } A \rightarrow Stream_P \text{ true } A \\ tail &: Stream_P \text{ true } A \rightarrow Stream_P \text{ false } A \\ forget &: Stream_P \text{ true } A \rightarrow Stream_P \text{ false } A \\ zipWith &: (A \rightarrow B \rightarrow C) \rightarrow Stream_P b A \rightarrow Stream_P b B \rightarrow Stream_P b C \end{aligned}$$

The type $Stream_P b A$ stands for streams generated in chunks of size (at least) one, where the first chunk is guaranteed to be non-empty if the index b is true. The constructor $[_]$ marks the end of a chunk. Note how the indices ensure that a finished chunk is always non-empty, and that *tail* may only be used to inspect the chunk currently being constructed. The constructor *forget* is used to “forget” that a chunk is already finished; *forget* represents the identity function. This constructor is used in the implementation of fib_P (an alternative would be to give *zipWith* a more general type):

$$\begin{aligned} fib_P &: Stream_P \text{ true } \mathbb{N} \\ fib_P &= 0 :: [\# (1 :: zipWith _+ _ (forget fib_P) (tail fib_P))] \end{aligned}$$

The implementation of $\llbracket _ \rrbracket_P$ for this language is very similar to that for the language in Section 7, so it is omitted here. For details of this implementation, the proof of correctness of fib_P , and the proof of uniqueness of solutions of the defining equation for fib_P , see Danielsson (2010a).

7 Other Chunk Sizes

The language of the previous section can be generalised to support other chunk sizes (Danielsson 2010a). Larger chunk sizes can provide interesting situations. Consider the following alternative definition of the function *map* from Section 5:

$$\begin{aligned} map_2 &: (A \rightarrow B) \rightarrow Stream A \rightarrow Stream B \\ map_2 f (x :: xs) &\text{ with } ^b xs \\ map_2 f (x :: xs) \mid y :: ys &= f x :: \# (f y :: \# map_2 f (^b ys)) \end{aligned}$$

One can show that *map* and map_2 are extensionally equal:

$$map \approx map_2 : (f : A \rightarrow B) \rightarrow (xs : Stream A) \rightarrow map f xs \approx map_2 f xs$$

However, assuming that pattern matching is “strict”, they are not interchangeable. The following definition of the stream of natural numbers is productive, albeit not guarded:

$$\begin{aligned} nats &: Stream \mathbb{N} \\ nats &= 0 :: \# map suc nats \end{aligned}$$

The definition that we get by replacing *map* by *map₂*, on the other hand, is not productive:

$$\begin{aligned} nats_2 &: Stream \mathbb{N} \\ nats_2 &= 0 :: \# map_2 \text{ suc } nats_2 \end{aligned}$$

The first element of *nats₂* is 0, but *map₂* needs to access the first *two* elements of its argument stream in order to output anything.

We can perhaps get a better picture of the situation above using the following language:

data *Stream_P* (*m* : \mathbb{N}) : $\mathbb{N} \rightarrow Set \rightarrow Set_1$ **where**
 $[-] : \infty (Stream_P m m A) \rightarrow Stream_P m 0 A$
 $_{::} : A \rightarrow Stream_P m n A \rightarrow Stream_P m (\text{suc } n) A$
 $map : (A \rightarrow B) \rightarrow Stream_P m n A \rightarrow Stream_P m n B$

Stream_P m n A is a language of programs which generate streams of *As* in chunks of size *m*, where the first chunk has size *n*. We can define WHNFs and the *whnf* function as follows:

data *Stream_W* (*m* : \mathbb{N}) : $\mathbb{N} \rightarrow Set \rightarrow Set_1$ **where**
 $[-] : Stream_P m m A \rightarrow Stream_W m 0 A$
 $_{::} : A \rightarrow Stream_W m n A \rightarrow Stream_W m (\text{suc } n) A$
 $map_W : (A \rightarrow B) \rightarrow Stream_W m n A \rightarrow Stream_W m n B$
 $map_W f [xs] = [map f xs]$
 $map_W f (x :: xs) = f x :: map_W f xs$
 $whnf : Stream_P (\text{suc } m) n A \rightarrow Stream_W (\text{suc } m) n A$
 $whnf [xs] = [\text{b } xs]$
 $whnf (x :: xs) = x :: whnf xs$
 $whnf (map f xs) = map_W f (whnf xs)$

Stream programs where all chunks are non-empty can then be turned into streams using guarded core-cursion:

mutual
 $[-]_W : Stream_W (\text{suc } m) (\text{suc } n) A \rightarrow Stream A$
 $[x :: [xs]]_W = x :: \# [xs]_P$
 $[x :: (y :: xs)]_W = x :: \# [y :: xs]_W$
 $[-]_P : Stream_P (\text{suc } m) (\text{suc } n) A \rightarrow Stream A$
 $[xs]_P = [whnf xs]_W$

Using this language we cannot define *nats₂*. The following code is ill-typed:

$$\begin{aligned} nats_2 &: Stream_P 2 1 \mathbb{N} \\ nats_2 &= 0 :: [\# map \text{ suc } nats_2] \end{aligned}$$

On the other hand, the following definitions are accepted:

$$\begin{aligned} nats &: Stream_P 1 1 \mathbb{N} & nats'_2 &: Stream_P 2 2 \mathbb{N} \\ nats &= 0 :: [\# map \text{ suc } nats] & nats'_2 &= 0 :: 1 :: [\# map \text{ suc } nats'_2] \end{aligned}$$

The language above uses constant chunk sizes (with the possible exception of the first chunk). If more flexibility is needed, then one can index programs by chunk sizes:

data <i>Chunks</i> : <i>Set</i> where <i>next</i> : <i>Chunks</i> → <i>Chunks</i> <i>cons</i> : ∞ <i>Chunks</i> → <i>Chunks</i>	data <i>Stream_P</i> : <i>Chunks</i> → <i>Set</i> → <i>Set</i> ₁ where [₋] : ∞ (<i>Stream_P</i> <i>cs</i> <i>A</i>) → <i>Stream_P</i> (<i>next cs</i>) <i>A</i> _::_ : <i>A</i> → <i>Stream_P</i> (^b <i>cs</i>) <i>A</i> → <i>Stream_P</i> (<i>cons cs</i>) <i>A</i> ...
---	--

Here *Chunks* represents the chunk sizes used in the production of a stream: *next* stands for the start of a new chunk, and *cons* increases the size of the current chunk by one. Note that *next* is inductive and *cons* coinductive; this ensures that there are no infinite sequences of empty chunks.

Endrullis et al. (2010) point out that some approaches to productivity based on restricted forms of *moduli of production*—which are closely related to chunk sizes—cannot handle the following definition of the Thue-Morse sequence:

thue-morse : *Stream Bool*
thue-morse = false :: # (*map not* (*evens thue-morse*) ∇ *tail thue-morse*)

Here *evens xs* consists of every other element of *xs*, starting with the first, and *_∇_* interleaves two streams: $(x :: xs) \nabla ys = x :: \# (ys \nabla \sup b xs)$. This definition of *thue-morse* can be handled using programs indexed by *Chunks*; see Danielsson (2010a) for details.

8 Nested Applications

Before wrapping up, let us briefly consider nested applications of the function being defined, as in $\varphi (x :: xs) = x :: \# \varphi (\varphi xs)$. Definitions with nested applications are common when programs are written using continuation-passing style. To handle such applications one can include a constructor for the function in the type of programs:

data <i>Stream_P</i> (<i>A</i> : <i>Set</i>) : <i>Set</i> where _::_ : <i>A</i> → ∞ (<i>Stream_P</i> <i>A</i>) → <i>Stream_P</i> <i>A</i> φ_P : <i>Stream_P</i> <i>A</i> → <i>Stream_P</i> <i>A</i> data <i>Stream_W</i> (<i>A</i> : <i>Set</i>) : <i>Set</i> where _::_ : <i>A</i> → <i>Stream_P</i> <i>A</i> → <i>Stream_W</i> <i>A</i>	φ_W : <i>Stream_W</i> <i>A</i> → <i>Stream_W</i> <i>A</i> $\varphi_W (x :: xs) = x :: \varphi_P (\varphi_P xs)$ $whnf$: <i>Stream_P</i> <i>A</i> → <i>Stream_W</i> <i>A</i> $whnf (x :: xs) = x :: \sup b xs$ $whnf (\varphi_P xs) = \varphi_W (whnf xs)$
---	--

(The definition of $\llbracket - \rrbracket_P$ is omitted above.) By turning streams into programs one can then define φ :

$\llbracket - \rrbracket$: <i>Stream A</i> → <i>Stream_P</i> <i>A</i> $\llbracket x :: xs \rrbracket = x :: \# \llbracket \sup b xs \rrbracket$	φ : <i>Stream A</i> → <i>Stream A</i> $\varphi xs = \llbracket \varphi_P \llbracket xs \rrbracket \rrbracket_P$
---	--

In order to prove that φ satisfies its intended defining equation it can be helpful to use an equality proof language, as in Section 5, and to include a constructor for the congruence of φ_P in this language:

data \approx_{P-} : *Stream A* → *Stream A* → *Set* **where**
 ...
 $\varphi_P\text{-cong} : (xs\ ys : \text{Stream}_P\ A) \rightarrow \llbracket xs \rrbracket_P \approx_P \llbracket ys \rrbracket_P \rightarrow \llbracket \varphi_P xs \rrbracket_P \approx_P \llbracket \varphi_P ys \rrbracket_P$

For further details, see Danielsson (2010a), who also establishes that φ 's defining equation has a unique solution.

9 Related Work

This section is mainly concerned with discussing methods for establishing productivity *in systems based on guarded corecursion*. Other related work is discussed towards the end.

Rutten (2003) proves that certain operations on streams are well-defined by using a technique which is very similar to the one described in this paper. He defines a language E of real number stream expressions inductively (this language is similar to $\text{Stream}_{\mathbb{R}}$), and defines a stream coalgebra $c : E \rightarrow \mathbb{R} \times E$ by recursion over the structure of E (this corresponds to *whnf*). The type of streams is a final coalgebra, so from c one obtains a function of type $E \rightarrow \text{Stream } \mathbb{R}$ (corresponding to $\llbracket - \rrbracket_p$), which can be used to turn stream expressions into actual streams. Rutten then uses coinduction (expressed using bisimulations) to prove that the defined operations satisfy their intended defining equations, and that these equations have unique solutions.

There are some differences between Rutten's proof and the technique described here, other than the different settings (finality vs. guarded corecursion, bisimulations vs. guarded coinduction). One is that Rutten defines the variant of *fib* from Section 6 via two mutually recursive streams ($\text{fib} = 0 :: \# \text{fib}'$ and $\text{fib}' = 1 :: \# \text{zipWith } _ + _ \text{ fib fib}'$); he does not discuss anything resembling the counting approaches of Sections 6 and 7. Another difference is that Rutten's language E is inductive, whereas $\text{Stream}_{\mathbb{R}}$ uses mixed induction and coinduction. A simple consequence of this difference is that when Rutten defines *fib* he includes it as a term in E ; with the method described here one can get much further using a fixed language. Danielsson and Altenkirch (2010) also take advantage of this difference when proving that one subtyping relation is sound with respect to another. In this proof the program and WHNF types are defined mutually, using mixed induction and coinduction, and the *whnf* function constructs its result using a combination of structural recursion and guarded corecursion. For completeness a short variant of this development is included in Appendix A.

Rutten's proof is closely related to a technique due to Bartels (2003). Bartels formulates the technique in a general categorical setting, and restricts the form of *whnf*, and in return proofs showing that the definitions uniquely satisfy certain defining equations come for free. Furthermore Bartels manages to define *fib* without including it as a term in the language.

Niqui (2009, 2010) implements one of Bartels' corecursion schemes, λ -coiteration, in Coq. He states that this scheme cannot handle van de Snepscheut's corecursive definition of the Hamming numbers (Dijkstra 1981), which can easily be handled using the method described in this paper.

Matthews (1999) and Di Gianantonio and Miculan (2003) describe general frameworks for defining values using a mixture of recursion and corecursion, based on functions which satisfy notions of contractivity. The methods seem to be quite general, and have been implemented (in Isabelle and Coq, respectively; note that guarded corecursion is not a primitive feature of Isabelle).

The implementations mentioned above (Matthews 1999; Di Gianantonio and Miculan 2003; Niqui 2009, 2010) provide you with unique solutions to equations, whereas when using the method described in this paper you need to prove correctness and uniqueness manually if you are interested in these properties. On the other hand, as pointed out in Section 5, there is rarely any need to pay this price when defining a proof. I suspect that circumstances determine which method is cheapest to use.

Bertot (2005) implements a filter function for streams in Coq. An unrestricted filter function is not productive, so Bertot restricts the function's inputs using predicates of the form "always (eventually

$P)$ ”. The always part is defined coinductively, and the eventually part inductively. As mentioned in the introduction this work is orthogonal to the work presented here.

Conor McBride (personal communication) has developed a technique for establishing productivity, based on the work of Hancock and Setzer (2000). The idea is to represent the right-hand sides of function definitions using a type $RHS\ g$, where g indicates whether the context is guarding or not, and to only allow corecursive calls in a guarding context.

Capretta (2005) defines the partiality monad, which can be used to represent partial (potentially non-terminating) computations, roughly as follows:

```
data  $_^v$  ( $A : Set$ ) :  $Set$  where
  return :  $A \rightarrow A^v$ 
  step   :  $\infty(A^v) \rightarrow A^v$ 
```

The constructor `return` returns a result, and `step` postpones a computation. It is easy to define `bind` for this monad: $_>>=_ : A^v \rightarrow (A \rightarrow B^v) \rightarrow B^v$. Unfortunately it can be inconvenient to use this definition of `bind` in systems based on guarded corecursion, because $_>>=_$ is not a constructor. Megacz (2007) suggests (more or less) the following alternative definition:

```
data  $_^v$  ( $A : Set$ ) :  $Set_1$  where
  return :  $A \rightarrow A^v$ 
   $\_>>=_$  :  $\infty(B^v) \rightarrow (B \rightarrow \infty(A^v)) \rightarrow A^v$ 
```

One can note that this is very close to the first step of the technique presented in this paper. Megacz does not translate from the second to the first type, though.

Bertot and Komendantskaya (2009) describe a method for replacing corecursion with recursion. They map values of type *Stream* A to and from the isomorphic type $\mathbb{N} \rightarrow A$, and values of this type can be defined recursively. The authors state that the method is still very limited and that, as presented, it cannot handle van de Snepscheut’s definition of the Hamming numbers.

McBride (2009) defines an applicative functor which captures the notion of “be[ing] ready a wee bit later”. Using this structure he defines various corecursive programs, including the circular breadth-first labelling function which is defined in Section 4. The technique is presented using the partial language Haskell, but Robert Atkey (personal communication) has later implemented it in Agda. The technique has not been developed very far yet: as far as I am aware no one has tried to prove any properties about functions defined using it.

Instead of working around the limitations of guarded corecursion one can include language features which make it easier to explain why programs are productive. One such feature is *sized types* (Hughes et al. 1996; Barthe et al. 2004; Abel 2009), and the λ -calculi of Buchholz (2005) provide other examples. Another approach is to use cleverer algorithms for establishing productivity. Endrullis et al. (2010, 2008) present algorithms which handle the definition of *thue-morse* from Section 7 automatically (except that, as presented, they only support first-order term-rewriting systems). The algorithms are tailored to streams; it seems to be hard to adapt them to, say, coinductive trees. Another algorithm is presented by Telford and Turner (1997). This algorithm does not handle *thue-morse* (Endrullis et al. 2010), but has the advantage of working for a large class of coinductive data types.

Morris et al. (2006) use the technique of replacing functions with constructors to show *termination* rather than productivity (see Morris et al. (2007) for an explanation of the technique). They replace a partially applied recursive call (which is not necessarily structural, because it could later be applied

to anything), nested inside another recursive call, with a constructor application. If this constructor application is later encountered it is handled using structural recursion.

The technique presented here also shares some traits with Reynolds' *defunctionalisation* (1972). Defunctionalisation is used to translate programs written in higher-order languages to first-order languages, and it basically amounts to representing function spaces using application-specific data types, and implementing interpreters for these data types.

10 Conclusions

I hope to have shown, through a number of examples, that the language-based approach to establishing productivity is useful. I am currently turning to it whenever I have a problem with guardedness; see Danielsson and Altenkirch (2010) and Danielsson (2010b) for some examples not included in this paper.

However, there are some problems with the method. As discussed in Section 5 it is not very useful if efficiency is a concern. Furthermore it can be disruptive: if one decides to use the method after already having developed a large number of functions in some project, and many of these functions have to be reified as constructors in a program data type, then a lot of work may be necessary. In fact, this problem—in one shape or another—is likely to apply to *all* approaches to making definitions guarded. In the long term I believe that it would be useful to adopt a more modular approach to productivity than guardedness.

Acknowledgements. I would like to thank Andreas Abel, Thorsten Altenkirch, Conor McBride, Nicolas Oury and Anton Setzer for many discussions about coinduction, and Graham Hutton as well as several anonymous reviewers for useful feedback. I would also like to thank EPSRC for financial support (grant code: EP/E04350X/1).

A An Inductive Approximation of Stream Equality

Danielsson and Altenkirch (2010) prove that one subtyping relation is sound with respect to another using the technique described in this paper. This appendix outlines the proof, but in a simplified (and slightly different) setting: equality between streams.

Recall the definitions of *Stream* and stream equality, \approx , from Section 2. One can define a sound approximation of stream equality inductively as follows (using an idea due to Brandt and Henglein (1998)):

data $_ \vdash _ \approx _$ $(H : \text{List } (\text{Stream } A \times \text{Stream } A)) : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set}$ **where**
 $_ :: _ : (x : A) \rightarrow (x :: xs, x :: ys) :: H \vdash _ \approx _ \rightarrow H \vdash x :: xs \approx x :: ys$
 $\text{hyp} : (xs, ys) \in H \rightarrow H \vdash xs \approx ys$
 $\text{trans} : H \vdash xs \approx ys \rightarrow H \vdash ys \approx zs \rightarrow H \vdash xs \approx zs$

The intention is that, if one can prove $H \vdash xs \approx ys$, and all the assumptions in the list H are valid, then xs and ys should be equal. The first constructor of $_ \vdash _ \approx _$ states that, in order to prove that $x :: xs$ and $x :: ys$ are equal, it suffices to show that $_ \vdash xs \approx _ \vdash ys$ are equal, given the extra assumption that $x :: xs$ and $x :: ys$ are equal. The second constructor makes it possible to use the hypotheses in the list H ($_ \in _$ encodes list membership), and the third constructor encodes transitivity. As an example, we can prove that the list $\text{repeat } x \approx x :: \# \text{repeat } x$ is equal to itself as follows:

$repeat_refl : (x : A) \rightarrow [] \vdash repeat\ x \approx repeat\ x$
 $repeat_refl\ x = x :: hyp\ here$

(The constructor here proves that the head of a list is a member of the list. In this case it is used at the type $(repeat\ x, repeat\ x) \in (repeat\ x, repeat\ x) :: []$.)

Soundness of $_ \vdash _ \approx _$ will now be established. The goal is to prove that $H \vdash xs \approx ys$ implies $xs \approx ys$, given that $All\ (Valid\ _ \approx _) H$, where $All\ P\ xs$ means that P holds for all elements in the list xs , and $Valid$ is *uncurry* for stream predicates:

data $All\ (P : A \rightarrow Set) : List\ A \rightarrow Set$ **where**
 $[] : All\ P\ []$
 $_ :: _ : P\ x \rightarrow All\ P\ xs \rightarrow All\ P\ (x :: xs)$
 $Valid : (Stream\ A \rightarrow Stream\ A \rightarrow Set) \rightarrow Stream\ A \times Stream\ A \rightarrow Set$
 $Valid\ _ R _ (xs, ys) = xs\ R\ ys$

We begin by defining the program and WHNF types mutually as follows:

mutual
data $_ \approx_P _ : Stream\ A \rightarrow Stream\ A \rightarrow Set$ **where**
 $sound : All\ (Valid\ _ \approx_W _) H \rightarrow H \vdash xs \approx ys \rightarrow xs \approx_P ys$
 $trans : xs \approx_P ys \rightarrow ys \approx_P zs \rightarrow xs \approx_P zs$
data $_ \approx_W _ : Stream\ A \rightarrow Stream\ A \rightarrow Set$ **where**
 $_ :: _ : (x : A) \rightarrow \infty\ (^b xs \approx_P ^b ys) \rightarrow x :: xs \approx_W x :: ys$

Note that the first argument of the program $sound$ refers to WHNFs. The function $trans_W$, whose type is $xs \approx_W ys \rightarrow ys \approx_W zs \rightarrow xs \approx_W zs$, can be defined using simple case analysis. The function $sound_W$ is defined as follows, using structural recursion:

$sound_W : All\ (Valid\ _ \approx_W _) H \rightarrow H \vdash xs \approx ys \rightarrow xs \approx_W ys$
 $sound_W\ valid\ (hyp\ h) = lookup\ valid\ h$
 $sound_W\ valid\ (trans\ xs \approx_P ys\ ys \approx_P zs) = trans_W\ (sound_W\ valid\ xs \approx_P ys)\ (sound_W\ valid\ ys \approx_P zs)$
 $sound_W\ valid\ (x :: xs \approx_P ys) = proof$
where $proof = x :: \# sound\ (proof :: valid)\ xs \approx_P ys$

In the first clause $lookup : All\ P\ xs \rightarrow x \in xs \rightarrow P\ x$ is used to fetch a proof from the “list” of valid assumptions. In the third clause a *circular* proof is constructed using guarded corecursion; note that the list of valid assumptions is extended with the proof currently being defined. Given $trans_W$ and $sound_W$ it is easy to define $whnf$ using structural recursion:

$whnf : xs \approx_P ys \rightarrow xs \approx_W ys$
 $whnf\ (sound\ valid\ xs \approx_P ys) = sound_W\ valid\ xs \approx_P ys$
 $whnf\ (trans\ xs \approx_P ys\ ys \approx_P zs) = trans_W\ (whnf\ xs \approx_P ys)\ (whnf\ ys \approx_P zs)$

The remaining pieces of the soundness proof are omitted (see Danielsson (2010a)).

References

Andreas Abel. Mixed inductive/coinductive types and strong normalization. In *APLAS 2007*, volume 4807 of *LNCS*, pages 286–301, 2009. doi:10.1007/978-3-540-76637-7_19.

- The Agda Team. The Agda Wiki. Available at <http://wiki.portal.chalmers.se/agda/>, 2010.
- Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. Note supporting presentation given at the Workshop on Partiality and Recursion in Interactive Theorem Provers, Edinburgh, UK, 2010.
- Falk Bartels. Generalised coinduction. *Math. Struct. in Comp. Science*, 13(2):321–348, 2003. doi:10.1017/S0960129502003900.
- G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comp. Science*, 14(1):97–141, 2004. doi:10.1017/S0960129503004122.
- Yves Bertot. Filters on coinductive streams, an application to Eratosthenes’ sieve. In *TLCA 2005*, volume 3461 of *LNCS*, pages 102–115, 2005. doi:10.1007/11417170_9.
- Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In *TYPES 2008*, volume 5497 of *LNCS*, pages 220–236, 2009. doi:10.1007/978-3-642-02444-3_14.
- Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.
- Wilfried Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Annals of Pure and Applied Logic*, 136(1–2):75–90, 2005. doi:10.1016/j.apal.2005.05.006.
- Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, 2005. doi:10.2168/LMCS-1(2:1)2005.
- Thierry Coquand. Infinite objects in type theory. In *TYPES’93*, volume 806 of *LNCS*, pages 62–78, 1994. doi:10.1007/3-540-58085-9_72.
- Nils Anders Danielsson. Code accompanying the paper. Available from the paper’s official download page, 2010a.
- Nils Anders Danielsson. Total parser combinators. In *ICFP’10*, pages 285–296, 2010b. doi:10.1145/1863543.1863585.
- Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively: An exercise in mixed induction and coinduction. In *MPC 2010*, volume 6120 of *LNCS*, pages 100–118, 2010. doi:10.1007/978-3-642-13321-3_8.
- Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In *TYPES 2002*, volume 2646 of *LNCS*, pages 148–161, 2003. doi:10.1007/3-540-39185-1_9.
- Edsger W. Dijkstra. Hamming’s exercise in SASL. EWD792 (privately circulated note), 1981.
- Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Data-oblivious stream productivity. In *LPAR 2008*, volume 5330 of *LNCS*, pages 79–96, 2008. doi:10.1007/978-3-540-89439-1_6.
- Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. *Theoretical Computer Science*, 411(4–5):765–782, 2010. doi:10.1016/j.tcs.2009.10.014.
- Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *CSL 2000*, volume 1862 of *LNCS*, pages 317–331, 2000. doi:10.1007/3-540-44622-2_21.
- Peter Hancock, Dirk Pattinson, and Neil Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3:9), 2009. doi:10.2168/LMCS-5(3:9)2009.

- Ralf Hinze. Functional pearl: Streams and unique fixed points. In *ICFP'08*, pages 189–200, 2008. doi:10.1145/1411204.1411232.
- John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423, 1996. doi:10.1145/237721.240882.
- Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report 071, Department of Computer Science, The University of Auckland, 1993.
- John Matthews. Recursive function definition over coinductive types. In *TPHOLs '99*, volume 1690 of *LNCS*, pages 73–90, 1999. doi:10.1007/3-540-48256-3_6.
- Conor McBride. Time flies like an applicative functor. Available at <http://www.e-pig.org/epilogue/?p=186>, 2009.
- Adam Megacz. A coinductive monad for Prop-bounded recursion. In *PLPV'07*, pages 11–20, 2007. doi:10.1145/1292597.1292601.
- Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *TYPES 2004*, volume 3839 of *LNCS*, pages 252–267, 2006. doi:10.1007/11617990_16.
- Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *Theory of Computing 2007, Proceedings of the Thirteenth Computing: The Australasian Theory Symposium (CATS2007)*, pages 111–121, 2007.
- Milad Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ -coiteration scheme. In *TYPES 2008*, volume 5497 of *LNCS*, pages 272–288, 2009. doi:10.1007/978-3-642-02444-3_17.
- Milad Niqui. Coiterative morphisms: Interactive equational reasoning for bisimulation, using coalgebras. Technical Report SEN-1003, Centrum Wiskunde & Informatica, 2010.
- Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.
- John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72*, volume 2, pages 717–740, 1972. doi:10.1145/800194.805852.
- J.J.M.M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1–3):1–53, 2003. doi:10.1016/S0304-3975(02)00895-2.
- Ben A. Sijtsma. On the productivity of recursive list definitions. *TOPLAS*, 11(4):633–649, 1989. doi:10.1145/69558.69563.
- Alastair Telford and David Turner. Ensuring streams flow. In *AMAST'97*, volume 1349 of *LNCS*, pages 509–523, 1997. doi:10.1007/BFb0000493.
- Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.